Extending the Built-In Solver

A new project automatically includes its own module that is loaded into the EUROPA engine and stub C++ files to hold custom code (see makeproject). Therefore, extending EUROPA components simply requires C++ code be added to those existing files. Consider the Example project here, which can be downloaded with:

```
svn co http://europa-pso.googlecode.com/svn/benchmarks/trunk/Example  Example
```

Adding a Flaw Filter

Flaw filters determine what flaws shouldn't be considered by the solver. They use the solver matching rule syntax as well as any additional information contained in the planner configuration XML.

In order to create a new flaw filter, you must subclass the FlawFilter class and override the `FlawFilter::test` function. Because FlawFilter instances are created via a factory, the only argument your filter's constructor can take is a `const TiXmlElement&`, which will contain all of the available configuration information from the planner configuration file. This XML data must be passed on to the base FlawFilter constructor along with a boolean flag indicating whether or not it's "dynamic"--the return value of `FlawFilter::test` may change during search.

Perhaps counterintuitively, the `FlawFilter::test` function should return true for any Entity to be filtered *out*. One way to think of this is that the function returns true if the Entity is "caught" by the filter.

Before it can be used in the planner configuration file, your filter must be registered as described in the Component Registration section.

Adding a Decision Point

Decision points manage the set of available choices for an individual flaw, the order of those choices, what the current choice is, and do the work of acting on the plan database when the choice is made and retracted. A decision point is created only when the solver actually picks a flaw to try to resolve, is revisited when the solver backtracks to that flaw, and is deleted when the solver backtracks past that flaw. If the flaw is revisited, a new decision point is created--the old one isn't reused.

If all that's wanted is some additional behavior beyond what the three basic Europa decision points (unbound variable, threat, and open condition) do, it's best to sub-class the decision point you want and override the parts that you want. If you want to:

-Change the choice order

- Override the constructor to do so "eagerly", by changing the actual order of the list of choices, if it's computed in the constructor.
- Override the handleInitialize function, to change the way the list of choices is computed.
- Override the getNext function to do so "lazily", by changing the way each next choice is gotten.

-Change the actions performed to make/un-make a decision

Override the handleExecute and/or handleUndo functions.

-Change when the choices are exhausted

Override the hasNext method.

If you're trying to write a decision point to resolve an entirely new kind of flaw, first read the section Adding a Flaw Manager for information about writing a class to detect the new flaw type. Once you have a manager that detects the flaws, you'll have to create a sub-class of the base SolverDecisionPoint class and provide the needed behaviors in the overrides described above.

Decision points are instantiated by flaw handlers, and so don't have to be registered like the other Solver components.

Adding a Flaw Handler

Flaw handlers are used to determine the priority of a flaw and which kind of decision point should be created for it. The default flaw manager is a templatized class with the parameter being the decision point type to be instantiated when a match is found, and it defers to a static function in that class to perform custom matching, so flaw handlers are typically only extended to implement new decision-ordering schemes.

The FlawHandler class has a virtual fuction getPriority for determining the priority of a flaw involving a particular Entity, where flaws with lower priority values are resolved before those with higher values.

Before it can be used in the planner configuration file, your flaw handler must be registered as described in the Component Registration section.

Adding a Flaw Manager

A flaw manager watches the plan database for the presence of a particular kind of flaw, apply flaw filters to determine if that flaw is within the scope of the manager, and try to match the flaw up with a flaw handler for eventual resolution.

In order to implement managers for new kinds of flaws, the base FlawManager has methods that respond to addition, removal, and change events from the constraint engine and the plan database. Any new flaw must be related to one of these events. Note that the "notifyRemoved" events do some work to clean up matches on entities that are being removed, and so any overriding function must delegate to them.

Filtering and matching is done via the `FlawManager::staticMatch` and `FlawManager::dynamicMatch` functions. In the base class, these use the standard Solver matching methods, but sub-classes generally layer additional matching on top of these. For example, the ThreatManager adds checking to determine if the potential flaw is a token as well as delegating to the base function.

Before it can be used in the planner configuration file, your flaw manager must be registered as described in the Component Registration section.

Component Registration

The configuration model outlined here requires registration of concrete factories to link logical component names with actual C++ classes. This should be done in the `initialize()` method in the application's module. A macro

is provided to make this quite straightforward. An example fragment of C++ code to register required components is provided below:

```
REGSITER_COMPONENT(``VariableFlawManager'',
                   EUROPA::VariableFlawManager);
REGSITER_COMPONENT(``StaticFilter'',
                   EUROPA::EUROPA::StaticFilter);
REGSITER_COMPONENT(``DynamicFilter'',
                   EUROPA::DynamicFilter);
REGSITER_COMPONENT(``InfiniteOrDynamic'',
                   EUROPA::InfiniteOrDynamicFilterCondition);
REGSITER_COMPONENT(``MinValue'',
                   EUROPA::MinValueHeuristic);
REGSITER_COMPONENT(``MaxValue'',
                   EUROPA::MaxValueHeuristic);
REGSITER_COMPONENT(``RandomValue'',
                   EUROPA::MonteCarloSelector);
```

TODO: Point to a specific example where we do this.